A Kernel Specification Formalism with Higher-Order Parameterisation

A kernel specification formalism with higher-order parameterisation*

Donald Sannella† Andrzej Tarlecki‡

Abstract

A specification formalism with parameterisation of an arbitrary order is presented. It is given a denotational-style semantics, accompanied by an inference system for proving that an object satisfies a specification. The inference system incorporates, but is not limited to, a clearly identified type-checking component.

Special effort is made to carefully distinguish between parameterised specifications, which denote functions yielding classes of objects, and specifications of parameterised objects, which denote classes of functions yielding objects. To deal with both of these in a uniform framework, it was convenient to view specifications, which specify objects, as objects themselves, and to introduce a notion of a specification of specifications.

The formalism includes the basic specification-building operations of the ASL specification language. This choice, however, is orthogonal to the new ideas presented. The formalism is also institution-independent, although this issue is not explicitly discussed at any length here.

1 Introduction

The most basic assumption of work on algebraic specification is that software systems are modelled as algebras. The signature of the algebra gives the names of data types and of operations, and the algebra itself gives the semantics of the particular realisations of these data types and operations in the system. Consequently, to specify a software system viewed in this way means to give a signature and define a class of algebras over this signature, that is, describe a class of admissible realisations of the types and operations.

The standard way to give a specification of a system in work on algebraic specification is to present a list of axioms over a given signature and describe in this way the properties that the operations of the system are to satisfy. This view of algebraic specification is perhaps the simplest possible, but has a number of disadvantages. Most notably, any specification of a real software system given in this style would comprise a very long, unstructured, and hence unmanageable list of axioms.

An obvious solution to this problem is to devise a *specification language* to build specifications in a structured fashion, using some *specification-building operations* to form complex specifications by putting together smaller and presumably well-understood pieces. The need for structure in specifications is universally recognized, and mechanisms for structuring specifications appear in all modern algebraic specification languages including CLEAR [BG 80], CIP-L [Bau 85], ASL [SW 83], [Wir 86], ACT ONE [EM 85], PLUSS [BGM 89] and the Larch Shared Language [GHW 85].

An important structuring mechanism is *parameterisation*. A parameterised specification P may be applied to any non-parameterised specification SP_{arg} fitting a certain signature Σ_{par} (or parameter specification SP_{par}) to yield a specification $P(SP_{arg})$. Hence, parameterised specifications are transformations mapping (argument) specifications to (result) specifications. A standard example is a specification *Stack-of-X* which takes a specification of stack elements and produces a specification of stacks

*Much of the material presented here has been included in a very preliminary form in Section 6 of [SST 90].

†LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland.

‡Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

tion mechanism, although the exact technicalities vary considerably[1]. In some algebraic specification frameworks, parameterisation is implicit in the sense that no distinction is made between parameterised and non-parameterised specifications (see for example LOOK [ETLZ 82], ASPIK [Voß 85] and the unified algebra framework [Mos 89a], [Mos 89b]) but the idea is the same.

Quite similarly, adequate structuring mechanisms are needed to organise programs to facilitate their development and understanding (and to enable separate compilation of program components). Many modern programming languages, beginning with Simula [DMN 70] and including Modula-2 [Wirth 88], CLU [Lis 81], Ada [Ada 80] and Standard ML [MTH 90] provide some notion of a program module to allow the programmer to structure the code being written. Again, an important structuring mechanism here is parameterisation. A parameterised program module F (an ML functor [MacQ 86], cf. [Gog 84]) may be applied to any non-parameterised program module A_{arg} matching a given import interface A_{par}. The result is a non-parameterised program module $F(A_{arg})$, a version of F in which the types and functions in A_{par} have been instantiated to the matching types and functions in A_{arg}. An example of a parameterised program module is a parser module which takes a lexical analyser module as argument. Since we model programs as algebras, such parameterised program modules are naturally modelled as functions mapping (argument) algebras to (result) algebras, i.e., algebras parameterised by other algebras. Somewhat informally, we will refer to such objects as *parametric algebras* (cf. *algebra modules* in OBSCURE [LL 88]). It is important to realise that such parametric algebras model self-contained programming units, and hence may correspond to independent programming tasks in the process of development of a software system.

A common drawback of the specification languages mentioned above is that they are predominantly concerned with specifications of non-parametric algebras without any provision for the structuring mechanisms used to construct complex programs (algebras) in a modular way. In particular, they do not provide any explicit concept of a specification of parametric algebras. In some specification frameworks this comes in, but only implicitly as an alternative interpretation of the concept of parameterised specification used in the formalism. For example, the "parameterised specifications" of ACT ONE [EM 85] are interpreted both as means of transforming specifications, i.e., parameterised specifications in our sense, and as a description of a certain functor on algebras, i.e., of a parametric algebra in our sense. Unfortunately, this dual view of "parameterised specifications" imposes in effect a requirement that the structure of a program implementing a specification, composed of (possibly parametric) algebras, must follow the structure of the specification, composed of (possibly parameterised) specifications. This not only violates the principle that a requirements specification is to describe the *what* without indicating the *how* of the system, but also is not acceptable from a practical point of view (see [FJ 90] for a realistic example of a specification with a structure entirely different from the structure of a software system it describes). We have discussed this issue in much detail in [SST 90], where our conclusion was summarised by the following slogan:

$$\text{parameterised (program specification)} \neq \text{(parameterised program) specification}$$

In short, we want a specification language where one can formulate both parameterised specifications on one hand and specifications of parameterised programs on the other.

Another idea for which we have argued in [SST 90] is an extensive use of higher-order parameterisation. Higher-order parameterisation arises not only because higher-order parametric algebras and their specifications are natural to consider from the semantic point of view, but more importantly because they are desirable from the methodological point of view: the use of higher-order parameterisation gives more flexibility in the process of systematic software development. In our opinion, this issue again has not been given proper attention in the specification languages mentioned above.

[1] In particular, the phrase "parameterised specification" has been reserved in some work on algebraic specification (see e.g. [Ehr 82] or [EKTWW 84]) for a formal object (a pair of specifications) which determines a parameterised specification in our sense via so called "parameter passing". We could not think of a better phrase to name "specifications that are parameterised by other specifications", hence the terminological clash.

example, such a possibility exists in COLD–K [FJKR 87] and has been considered for ASL [SW 83], [ST 88]). We believe that all the benefits of higher-order parameterisation come to light only in the context of a careful distinction between parameterised specifications and specifications of parametric algebras.

In this paper we present our first attempt to incorporate the two methodological ideas sketched above into a specification language. We propose here a specification formalism which builds on the simple yet powerful specification-building operations of ASL (this choice is not essential for the development presented in this paper) and incorporates a parameterisation mechanism capable of describing parametric algebras of an arbitrary order and their specifications, as well as parameterised specifications of an arbitrary order. It was possible to use a single parameterisation mechanism in all these situations because our formalism gives arbitrary specifications the status of first-level objects. Thus, specifications which are primarily used to specify "simpler" objects of the language, are themselves treated also as objects, which in turn may be specified, passed as arguments to functions and arise as results of function application.

The parameterisation mechanism added is inspired by the λ-abstraction mechanism of typed λ-calculi (thus, it generalises the original parameterisation mechanism of ASL [SW 83], [ST 88]). It is important to realise that although the objects of the formalism we propose look like typed λ-expressions, the underlying intuition is slightly different. We like the phrase *specified λ-calculus* as a possible indication of the difference. In typed λ-calculi, the admissible arguments of a function defined by a λ-expression are described just by stating their required type; it is intuitively expected that it will be easy to determine statically whether or not application of such a function to an argument is well-formed. This is in contrast with the situation in specified λ-calculi such as the formalism we propose: the admissible arguments are specified here rather than just being characterised by a type, and so a full-blown verification process is required to determine well-formedness of application.

The paper is organised as follows. Section 2 lists the usual algebraic prerequisites we assume the reader to be familiar with and recalls, for the sake of completeness of the definitions given later, the specification-building operations of ASL. A brief informal description of the language we propose, including its syntax, is given in Section 3. A denotational-style semantics of the language is in Section 4. Section 5 studies the well-formedness and verification of the objects of the language. We point out that the two are necessarily intertwined, and present a formal system to derive judgements of the form $Obj : SP$ stating that an object Obj satisfies a specification SP. Some basic properties of the system are then proved in the second part of the section. Although it is impossible to determine well-formedness of objects of the language using purely "static" type-checking technology, the verification process as presented in Section 5 contains an intuitively clear type-checking component. We introduce a notion of type appropriate for our language in Section 6, and then use it to separate this "type-checking" component from the verification process. Finally, a summary of the topics presented in the paper and some discussion of directions for further work is given in Section 7.

2 Preliminaries

Throughout the paper we assume that the reader is familiar with the basic concepts of logic and universal algebra. In particular we will freely use the notions of: algebraic many-sorted signature, usually denoted by Σ, Σ', Σ_1, etc.; algebraic signature morphism $\sigma : \Sigma \to \Sigma'$; Σ-algebra; Σ-homomorphism; Σ-isomorphism; Σ-equation; first-order Σ-sentence (the set of all Σ-sentences will be denoted by $Sen(\Sigma)$); and satisfaction relation between Σ-algebras and Σ-sentences. These all have the usual definitions (see e.g. [ST 88]) and a standard, hopefully self-explanatory notation is used to write them down. We will also use the standard notation and concepts of λ-calculus, in particular, free and bound occurrences of variables, substitution, β-reduction etc., cf. [Bar 84].

For any signature Σ, the class of all Σ-algebras is denoted by $Alg(\Sigma)$. We will identify this with the category of Σ-algebras and Σ-homomorphisms whenever convenient. If $\sigma : \Sigma \to \Sigma'$ is a signature

$-|_\Sigma$ is sometimes used when σ is obvious).

The most essential feature of any specification formalism is that every specification SP over a given signature Σ (we will say that SP is a Σ-specification) unambiguously determines a class of Σ-algebras (sometimes referred to as *models* of the specification) $[\![SP]\!] \in Pow(Alg(\Sigma))$[2]. See [ST 88] for a more extensive discussion of the semantics of specifications.

As a starting point for the presentation of specifications in this paper, we recall here the simple yet powerful specification-building operations defined in [ST 88] (with the slight difference that signatures are regarded as specifications in their own right here with **impose** Φ **on** Σ in place of $\langle \Sigma, \Phi \rangle$). These were in turn based on the ASL specification language [SW 83], [Wir 86]. Even though the particular choice of specification-building operations is not important for the purposes of this paper, we give here their full formal definitions to make the paper self-contained. We refer the reader to [ST 88] for a full explanation of the motivation, intuitive understanding and technical machinery behind these definitions.

- If Σ is a signature, then Σ is a Σ-specification with the semantics:

$$[\![\Sigma]\!] = Alg(\Sigma)$$

- If SP is a Σ-specification and Φ is a set of Σ-sentences, then **impose** Φ **on** SP is a Σ-specification with the semantics:

$$[\![\textbf{impose } \Phi \textbf{ on } SP]\!] = \{A \in [\![SP]\!] \mid A \models \Phi\}$$

- If SP is a Σ-specification and $\sigma : \Sigma' \to \Sigma$ is a signature morphism, then **derive from** SP **by** σ is a Σ'-specification with the semantics:

$$[\![\textbf{derive from } SP \textbf{ by } \sigma]\!] = \{A|_\sigma \mid A \in [\![SP]\!]\}$$

- If SP is a Σ-specification and $\sigma : \Sigma \to \Sigma'$ is a signature morphism, then **translate** SP **by** σ is a Σ'-specification with the semantics:

$$[\![\textbf{translate } SP \textbf{ by } \sigma]\!] = \{A' \in Alg(\Sigma') \mid A'|_\sigma \in [\![SP]\!]\}$$

- If SP and SP' are Σ-specifications, then $SP \cup SP'$ is a Σ-specification with the semantics:

$$[\![SP \cup SP']\!] = [\![SP]\!] \cap [\![SP']\!]$$

- If SP is a Σ-specification and $\sigma : \Sigma' \to \Sigma$ is a signature morphism, then **minimal** SP **wrt** σ is a Σ-specification with the semantics:

$$[\![\textbf{minimal } SP \textbf{ wrt } \sigma]\!] = \{A \in [\![SP]\!] \mid A \text{ is minimal in } Alg(\Sigma) \text{ w.r.t. } \sigma\}$$[3]

  where a Σ-algebra A is *minimal w.r.t.* σ if it has no non-trivial subalgebra with an isomorphic σ-reduct (cf. [ST 88]).

- If SP is a Σ-specification, then **iso-close** SP is a Σ-specification with the semantics:

$$[\![\textbf{iso-close } SP]\!] = \{A \in Alg(\Sigma) \mid A \text{ is isomorphic to } B \text{ for some } B \in [\![SP]\!]\}$$

[2] $Pow(X)$, for any class X, denotes the "class of all subclasses" of X. This raises obvious foundational difficulties. We disregard these here, as they may be resolved in a number of standard ways. For example, for the purposes of this paper we could assume that algebras are built within an appropriate universal set, and deal with sets, rather than classes, of algebras.

then **abstract** SP **wrt** Φ' **via** σ is a Σ-specification with the semantics:

$$[\![\textbf{abstract } SP \textbf{ wrt } \Phi' \textbf{ via } \sigma]\!] = \{A \in Alg(\Sigma) \mid A \equiv^{\sigma}_{\Phi'} B \text{ for some } B \in [\![SP]\!]\}$$

where $A \equiv^{\sigma}_{\Phi'} B$ means that A is observationally equivalent to B w.r.t. Φ' via σ. The concept of observational equivalence used here covers as special cases the different notions of behavioural equivalence with respect to a set of observable sorts which appear in the literature. The set Φ' contains formulae over Σ (with "free variables" introduced by σ) intended to characterise the relevant aspects of the "behaviour" of Σ-algebras. If no free variables are involved (σ is the identity morphism on Σ) then $A \equiv^{\sigma}_{\Phi'} B$ holds iff A and B satisfy exactly the same sentences from Φ'. (See [ST 87], [ST 88] for details.)

The above definitions were given in [ST 88] in the framework of an arbitrary *institution* [GB 84]. This means that the specification-building operations defined above are actually independent of the underlying logical system, that is, of the particular definitions of the basic notions of signature, algebra, sentence and satisfaction relation. This is an important advantage: we can use the operations in an arbitrary logical system (formalised as an institution) without having to redefine them each time we decide to modify the underlying notions; see [GB 84] and [ST 88] for a discussion of this issue.

3 Introducing the language

The specification formalism we develop in this paper extends in an essential way the kernel specification language presented in [ST 88] by adding a simple yet powerful parameterisation mechanism which allows us to define and specify parametric algebras of arbitrary order, as well as extending the mechanism in [ST 88] for defining first-order parameterised specifications to the higher-order case. This is achieved by viewing specifications on one hand as specifications of objects such as algebras or parametric algebras, and on the other hand as objects themselves to which functions (i.e. parameterised specifications) may be applied. Consequently, the language allows specifications to be specified by other specifications, much as in CLEAR [BG 80] or ACT ONE [EM 85] parameterisation where the parameter specification specifies the permissible argument specifications.

The view of specifications as objects enables the use of a uniform parameterisation mechanism, functions defined by means of λ-abstraction, to express both parameterised specifications and parametric algebras. There is also a uniform specification mechanism to specify such functions, Π-abstraction (Cartesian-product specification, closely related to the dependent function type constructor in e.g. NuPRL [Con 86]). This may be used to specify (higher-order) parametric algebras as well as (higher-order) parameterised specifications. There is no strict separation between levels, which means that it is possible to intermix parameterisation of objects and parameterisation of specifications, obtaining (for example) algebras which are parametric on parameterised specifications or specifications which are parameterised by parametric algebras. We have not yet explored the practical implications of this technically natural generalisation.

The language does not include notation for describing algebras, signatures, signature morphisms, or sets of sentences. Such notation must be provided separately, for example as done for ASL in [Wir 86]. The definition of the language is independent of this notation; moreover, it is essentially *institution independent*, with all the advantages indicated in [GB 84], [ST 88].

The language has just one syntactic category of interest, which includes both specifications and

[3] This is slightly different from the definition in [ST 88].

$$
\begin{array}{lll}
Object & = & \left.\begin{array}{l}
Signature \\
\mid\ \textbf{impose}\ Sentences\ \textbf{on}\ Object \\
\mid\ \textbf{derive from}\ Object\ \textbf{by}\ Signature\text{-}morphism \\
\mid\ \textbf{translate}\ Object\ \textbf{by}\ Signature\text{-}morphism \\
\mid\ Object \cup Object \\
\mid\ \textbf{minimal}\ Object\ \textbf{wrt}\ Signature\text{-}morphism \\
\mid\ \textbf{iso-close}\ Object \\
\mid\ \textbf{abstract}\ Object\ \textbf{wrt}\ Sentences\ \textbf{via}\ Signature\text{-}morphism
\end{array}\right\} \text{Simple specifications} \\
 & & \left.\begin{array}{l}
\mid\ \Pi\, Variable{:}Object.\ Object \\
\mid\ \{Object\} \\
\mid\ \textbf{Spec}(Object)
\end{array}\right\} \text{Other specifications} \\
 & & \left.\begin{array}{l}
\mid\ Variable \\
\mid\ Algebra\text{-}expression \\
\mid\ \lambda\, Variable{:}Object.\ Object \\
\mid\ Object(Object)
\end{array}\right\} \text{Other objects}
\end{array}
$$

As usual, we have omitted the "syntax" of variables. The other syntactic categories of the language above are algebra expressions, signatures, sets of sentences and signature morphisms — as mentioned above, the details of these are not essential to the main ideas of this paper and we assume that they are provided externally. Algebra expressions may contain occurrences of object variables. We will assume, however, that variables do not occur in signatures, signature morphisms and sentences, which seems necessary to keep the formalism institution-independent. This requirement may seem overly restrictive, as it seems to disallow the components of a particular algebra to be used in axioms; one would expect to be able to write something like $\Pi X{:}\Sigma.\,(\ldots X.op\ldots)$. Fortunately, using the power of the specification-building operations included in the language, it is possible to define a more convenient notation which circumvents this restriction (see Appendix A in [SST 90]).

We have used the standard notation for Π- and λ-objects to suggest the usual notions of a free and of a bound occurrence of a variable in a term of the language, as well as of a closed term. As usual, we identify terms which differ only in their choice of bound variable names. We define substitution of objects for variables in the usual way: $Obj[Obj'/X]$ stands for the result of substituting Obj' for all free occurrences of X in Obj in such a way that no unintended clashes of variable names take place. This also defines the usual notion of β-reduction between objects of the language: $(\ldots(\lambda X{:}SP.\,Obj)(Obj')\ldots) \rightarrow_\beta (\ldots Obj[Obj'/X]\ldots)$. Then, $\rightarrow_\beta^*$ is the reflexive and transitive closure of $\rightarrow_\beta$.

The first eight kinds of specifications listed above (simple specifications) are taken directly from [ST 88] (see Section 2). The particular choice of these eight operations is orthogonal to the rest of the language and will not interfere with the further development in this paper. The other three kinds of specifications are new. Π-abstraction is used to specify parametric objects. To make this work, it must be possible to use objects in specifications. The $\{\_\}$ operation provides this possibility by allowing objects to be turned into (very tight) specifications. The next clause allows a specification which defines a class $\mathcal{C}$ of objects to be turned into a specification which defines the class of specifications defining subclasses of $\mathcal{C}$. This is compatible with the use of parameter specifications in parameterised specifications as in CLEAR and ACT ONE. For example, the declaration **proc** $P(X : SP) = \ldots$ in CLEAR introduces a parameterised specification P, where the parameter (or *requirement*) specification SP describes the admissible arguments of P. Namely, if SP defines a class of objects $C = [\![SP]\!]$ then P may be applied to argument specifications SP_{arg} defining a subclass of C, i.e. such that $[\![SP_{arg}]\!] \subseteq [\![SP]\!]$ (we disregard the parameter fitting mechanism). In our formalism this would be written as $P \equiv \lambda X{:}\mathbf{Spec}(SP).\ \ldots$.

The syntax of other objects is self-explanatory.

The richness of the language may lead to some difficulty in recognizing familiar concepts which appear here in a generalised form. The following comments might help to clarify matters:

algebras, then this specification is a specification in the usual sense.

- $\Pi X{:}\ldots.\ldots$ denotes a class of mappings from objects to objects. If these objects are algebras, then this is a class of parametric algebras, i.e. a specification of a parameterised program.

- $\lambda X{:}\ldots.\ldots$ denotes a mapping from objects to objects. If these objects are specifications in the usual sense, then this is a parameterised specification.

The semantics of the language, presented in the next section, gives more substance to the informal comments above concerning the intended denotations of certain phrases.

As pointed out above, we assume that the sublanguage of expressions defining algebras is to be supplied externally (with a corresponding semantics — see Section 4). Even under this assumption, it would be possible to include institution-independent mechanisms for building algebras from other algebras (amalgamation, reduct, free extension, etc.) in the language, which could lead to a powerful and uniform calculus of specified modular programs. This is an interesting possibility for future work but it is outside the scope of this paper.

4 Semantics

We have chosen the syntax for objects in the language so that their semantics should be intuitively clear. We formalise it by defining for any environment ρ, which assigns meanings to variables, a partial function $[\![\_]\!]\rho$ mapping an object Obj to its meaning $[\![Obj]\!]\rho$. It is defined below by structural induction on the syntax of objects. The use of the meta-variable SP instead of Obj in some places below is intended to be suggestive (of objects denoting object classes, used as specifications) but has no formal meaning. This convention will be used throughout the rest of the paper.

Simple specifications:

$[\![\Sigma]\!]\rho = Alg(\Sigma)$
$[\![\textbf{impose } \Phi \textbf{ on } SP]\!]\rho = \{A \in [\![SP]\!]\rho \mid A \models \Phi\}$
if $[\![SP]\!]\rho \subseteq Alg(\Sigma)$ and $\Phi \subseteq Sen(\Sigma)$ for some signature Σ
$[\![\textbf{derive from } SP \textbf{ by } \sigma]\!]\rho = \{A\big|_\sigma \mid A \in [\![SP]\!]\rho\}$
if $[\![SP]\!]\rho \subseteq Alg(\Sigma)$ and $\sigma : \Sigma' \to \Sigma$ is a signature morphism for some signatures Σ and Σ'
... similarly for the other forms, based on the semantics given in Section 2 ...

Other specifications:

$[\![\{Obj\}]\!]\rho = \{[\![Obj]\!]\rho\}$
if $[\![Obj]\!]\rho$ is defined
$[\![\Pi X{:}SP.\ SP']\!]\rho = \Pi_{v \in [\![SP]\!]\rho} [\![SP']\!]\rho[v/X]$[4,5]
if $[\![SP]\!]\rho$ is a class of values and for each $v \in [\![SP]\!]\rho$, $[\![SP']\!]\rho[v/X]$ is a class of values
$[\![\textbf{Spec}(SP)]\!]\rho = Pow([\![SP]\!]\rho)$
if $[\![SP]\!]\rho$ is a class of values

$[\![X]\!]\rho = \rho(X)$
$[\![A]\!]\rho = \ldots$ *assumed to be given externally* $\ldots$
$[\![\lambda X{:}SP.\, Obj]\!]\rho = \{\langle v \mapsto [\![Obj]\!]\rho[v/X]^5\rangle \mid v \in [\![SP]\!]\rho\}$
if $[\![SP]\!]\rho$ is a class of values and for each $v \in [\![SP]\!]\rho$, $[\![Obj]\!]\rho[v/X]$ is defined
$[\![Obj(Obj')]\!]\rho = [\![Obj]\!]\rho([\![Obj']\!]\rho)$
if $[\![Obj]\!]\rho$ is a function and $[\![Obj']\!]\rho$ is a value in the domain of this function

In the above definition, it is understood that a condition like "$[\![SP]\!]\rho \subseteq Alg(\Sigma)$" implicitly requires that $[\![SP]\!]\rho$ is defined. An object's meaning is undefined unless the side-condition of the appropriate definitional clause holds.

It is easy to see that the semantics of an object of the language depends only on the part of the environment which assigns meanings to variables which occur free in the object. In particular, the meaning of a closed object is independent from the environment. That is, for any closed object Obj and environments ρ and ρ', $[\![Obj]\!]\rho$ is defined if and only if $[\![Obj]\!]\rho'$ is defined and if they are defined then $[\![Obj]\!]\rho = [\![Obj]\!]\rho'$. This allows us to omit the environment when dealing with the semantics of closed objects and write simply $[\![Obj]\!]$ to stand for $[\![Obj]\!]\rho$ for any environment ρ whenever Obj is closed.

Of course, the above remark is true only provided that the sublanguage of algebra expressions and its semantics assumed to be given externally have this property. In the following, we will take this for granted. We will also assume that the sublanguage satisfies the following substitutivity property: for any algebra expression A, variable X and object Obj, for any environment ρ such that $v = [\![Obj]\!]\rho$ is defined, $[\![A[Obj/X]]\!]\rho$ is defined if and only if $[\![A]\!]\rho[v/X]$ is defined, and if they are defined then they are the same. This ensures that the following expected fact holds for our language (the standard proof by induction on the structure of objects is omitted):

Fact 4.1 For any objects Obj, Obj' and variable X, for any environment ρ such that $v' = [\![Obj']\!]\rho$ is defined, $[\![Obj[Obj'/X]]\!]\rho$ is defined if and only if $[\![Obj]\!]\rho[v'/X]$ is defined, and if they are defined then

$$[\![Obj[Obj'/X]]\!]\rho = [\![Obj]\!]\rho[v'/X]$$

□

Corollary 4.2 β-reduction preserves the meaning of objects. That is, for any objects Obj and Obj' such that $Obj \rightarrow^*_\beta Obj'$, for any environment ρ, if $[\![Obj]\!]\rho$ is defined then so is $[\![Obj']\!]\rho$ and $[\![Obj]\!]\rho = [\![Obj']\!]\rho$. □

The above semantics is overly permissive in comparison with the semantics given to simple specifications in Section 2 and [ST 88] in the sense that it assigns meanings to some specifications which would be considered ill-formed according to the definitions given there. This is caused by the "polymorphic" character of the empty class of algebras. For example, if SP is an inconsistent Σ-specification (i.e., assuming SP is closed, $[\![SP]\!] = \emptyset$) then **impose** Φ **on** SP has a well-defined meaning (the empty class of algebras) even if Φ is a set of sentences over a signature which is completely unrelated to Σ. Generalising the treatment in Section 2 in the present context is possible via the notion of type to be introduced in Section 6. However, the use of specifications (rather than signatures and types) to constrain formal parameters makes such a type system insufficiently descriptive to ensure well-formedness of specifications. For this, full-blown verification, rather than just type-checking, is required. We will discuss this issue in more detail in the following sections.

[4] Π on the right-hand side of this definition denotes the usual Cartesian product of an indexed family of sets. That is, $\Pi_{x \in S} C_x$ is the set of all functions with domain S mapping any $x \in S$ to an element of C_x.

[5] As usual, $\rho[v/X]$ is the environment which results from ρ by assigning v to the variable X (and leaving the values of other variables unchanged).

of values, the elements of which are assigned to objects of the language as their meanings. A naive attempt might have been as follows:

$$Values = Algebras \mid Pow(Values) \mid Values \nrightarrow Values$$

Clearly, this leads to serious foundational problems, as the recursive domain definition involves "heavy recursion" (cf. [BT 83]) and hence cannot have a set-theoretic solution (even assuming that we consider here a set *Algebras* of algebras built within a fixed universe). However, since the formalism we introduce is not intended to cater for any form of self application of functions or non-well-foundedness of sets, the equation above attempts to define a domain of values of objects which is undesirably rich. The well-formed[6] objects of the language can easily be seen to form a hierarchy indexed by "types" (see Section 6). Thus, we can define a corresponding cumulative hierarchy of sets of values, and then define the domain of the meanings of objects as the union of sets in the hierarchy, much in the style of [BKS 88] (see [BT 83] where the idea of using hierarchies of domains in denotational semantics is discussed in more detail). Another, less "constructive", possibility is to work within a fixed universal set of values of objects containing the "set" of all algebras [Coh 81].

5 Proving satisfaction

We are interested in determining whether or not given objects satisfy given specifications. We use the formal judgement $Obj : SP$ to express the assertion that a closed object Obj satisfies a closed specification SP, i.e. that $[\![Obj]\!] \in [\![SP]\!]$, and generalise it to $X_1 : SP_1, \ldots, X_n : SP_n \vdash Obj : SP$ stating the assertion that an object Obj satisfies a specification SP in the *context* $X_1 : SP_1, \ldots, X_n : SP_n$, i.e. under the assumption that objects $X_1, \ldots, X_n$ satisfy specifications $SP_1, \ldots, SP_n$, respectively. The inference rules listed below allow us to derive judgements of this general form. For the sake of clarity, though, we have decided to make contexts implicit in the rules and rely on the natural deduction mechanism of introducing and discharging assumptions (all of the form $X : SP$ here) to describe the appropriate context manipulation. For example, in (R2) below, $[X : SP]$ is an assumption which may be used to derive $SP' : \mathbf{Spec}(SP'')$, but is discharged when we apply the rule to derive its conclusion. Whenever necessary, we will use the phrase "the current context" to refer to the sequence of currently undischarged assumptions. We say that an environment ρ is *consistent* with a context $X_1 : SP_1, \ldots, X_n : SP_n$ if for $i = 1, \ldots, n$, $\rho(X_i) \in [\![SP_i]\!]\rho$.

Simple specifications:

$$\frac{\Sigma \text{ signature}}{\Sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{\mathbf{impose}\ \Phi\ \mathbf{on}\ SP : \mathbf{Spec}(\Sigma)}$$

$$\frac{SP : \mathbf{Spec}(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{derive\ from}\ SP\ \mathbf{by}\ \sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{translate}\ SP\ \mathbf{by}\ \sigma : \mathbf{Spec}(\Sigma')}$$

$$\frac{SP : \mathbf{Spec}(\Sigma) \qquad SP' : \mathbf{Spec}(\Sigma)}{SP \cup SP' : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \sigma : \Sigma' \to \Sigma}{\mathbf{minimal}\ SP\ \mathbf{wrt}\ \sigma : \mathbf{Spec}(\Sigma)}$$

$$\frac{SP : \mathbf{Spec}(\Sigma)}{\mathbf{iso\text{-}close}\ SP : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi' \subseteq Sen(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{abstract}\ SP\ \mathbf{wrt}\ \Phi'\ \mathbf{via}\ \sigma : \mathbf{Spec}(\Sigma)}$$

[6] An intuitive understanding of the notion of well-formedness involved is sufficient here (we hope) — we introduce it formally in Section 5.

$$\frac{Obj : SP}{\{Obj\} : \mathbf{Spec}(SP)} \tag{R1}$$

$$\frac{SP : \mathbf{Spec}(SP_{any}) \qquad \begin{matrix}[X : SP] \\ SP' : \mathbf{Spec}(SP'')\end{matrix}}{\Pi X{:}SP.\ SP' : \mathbf{Spec}(\Pi X{:}SP.\ SP'')} \tag{R2}$$

$$\frac{SP : \mathbf{Spec}(SP')}{\mathbf{Spec}(SP) : \mathbf{Spec}(\mathbf{Spec}(SP'))} \tag{R3}$$

λ-expressions:

$$\frac{SP : \mathbf{Spec}(SP_{any}) \qquad \begin{matrix}[X : SP] \\ Obj : SP'\end{matrix}}{\lambda X{:}SP.\, Obj\ :\ \Pi X{:}SP.\ SP'} \tag{R4}$$

$$\frac{Obj : \Pi X{:}SP.\ SP' \qquad Obj' : SP}{Obj(Obj') : SP'[Obj'/X]} \tag{R5}$$

$$\frac{Obj : SP \qquad SP \rightarrow^*_\beta SP'}{Obj : SP'} \tag{R6}$$

$$\frac{Obj : SP \qquad SP' : \mathbf{Spec}(SP_{any}) \qquad SP' \rightarrow^*_\beta SP}{Obj : SP'} \tag{R7}$$

Trivial inference:

$$\frac{Obj : SP_{any}}{Obj : \{Obj\}} \tag{R8}$$

"Cut"

$$\frac{Obj : SP \qquad SP : \mathbf{Spec}(SP')}{Obj : SP'} \tag{R9}$$

$$\frac{SP : \mathbf{Spec}(\Sigma) \qquad [\![A]\!]\rho \in [\![SP]\!]\rho \text{ for all } \rho \text{ consistent with the current context}}{A : SP} \quad \text{(R10)}$$

$$\frac{SP, SP' : \mathbf{Spec}(\Sigma) \qquad [\![SP]\!]\rho \subseteq [\![SP']\!]\rho \text{ for all } \rho \text{ consistent with the current context}}{SP : \mathbf{Spec}(SP')} \quad \text{(R11)}$$

Some of these rules involve judgements (Σ signature, $\Phi \subseteq Sen(\Sigma)$, $\sigma : \Sigma \to \Sigma'$) which are external to the above formal system. This is a natural consequence of the fact that the language does not include any syntax for signatures, sentences, etc. More significantly, there are two rules which involve model-theoretic judgements, referring to the semantics of objects given above.

Following the usual practice, in the sequel we will simply write "$Obj : SP$" meaning "$Obj : SP$ is derivable".

The rules labelled *Simple specifications* characterise the well-formedness of Σ-specifications built using the underlying specification-building operations included in the language. They directly incorporate the "syntactic" requirements of Section 2 on the use of these operations. Rules (R1), (R2) and (R3) play a similar role for the other specification-forming operations: singleton specification, Cartesian-product specification and $\mathbf{Spec}(\_)$, respectively. Notice, however, that their specifications are given here in a form which is as tight as possible. For example, for any $SP : \mathbf{Spec}(\Sigma)$ and $Obj : SP$, rule (R1) allows us to deduce $\{Obj\} : \mathbf{Spec}(SP)$ rather than just $\{Obj\} : \mathbf{Spec}(\Sigma)$.

The rules related to λ-expressions and their applications to arguments are quite straightforward. Rules (R4) and (R5) are the usual rules for λ-expression introduction and application, respectively. The assumption $SP : \mathbf{Spec}(SP_{any})$ in rule (R4) asserts the well-formedness of the specification SP (see also (R2), (R7), (R8)). Whenever the meta-variable SP_{any} is used below, it will play the same role as part of a well-formedness constraint. Notice that in order to prove $\lambda X{:}SP.\, Obj : \Pi X{:}SP.\, SP'$, we have to prove $Obj : SP'$ "schematically" for an arbitrary unknown $X : SP$, rather than for all values in the class $[\![SP]\!]\rho$ (for the appropriate environments ρ).

Rules (R6) and (R7) embody a part of the observation that β-reduction preserves the semantics of objects (Corollary 4.2). Rule (R6) allows for β-reduction and rule (R7) for well-formed β-expansion of specifications. A particular instance of the latter is

$$\frac{Obj' : SP'[Obj/X] \qquad (\lambda X{:}SP.\, SP')(Obj) : \mathbf{Spec}(SP_{any})}{Obj' : (\lambda X{:}SP.\, SP')(Obj)}$$

That is, in order to prove that an object satisfies a specification formed by applying a parameterised specification to an argument, it is sufficient to prove that the object satisfies the corresponding β-reduct.

However, we have not incorporated full β-equality into our system; rules (R6) and (R7) introduce it only for specifications. In particular, we have not included the following rule, which would allow well-formed β-expansion of objects:

$$\frac{Obj : SP \qquad Obj' : SP_{any} \qquad Obj' \to^*_\beta Obj}{Obj' : SP}$$

An instance of this would be:

$$\frac{Obj_1[Obj_2/X] : SP \qquad (\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP_{any}}{(\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP}$$

Hence, in order to prove that a structured object $(\lambda X{:}SP_2.\, Obj_1)(Obj_2)$ satisfies a specification SP, it would suffice to show that the object is well-formed and to prove that its β-reduct $Obj_1[Obj_2/X]$

of a program should follow the structure of the program, without any possibility of flattening it out. So, to prove $(\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP$ we have to find an appropriate specification for the parameterised program $\lambda X{:}SP_2.\, Obj_1$, say $\lambda X{:}SP_2.\, Obj_1 : \Pi X{:}SP_2.\, SP_1$ such that $SP_1[Obj_2/X] = SP$ (actually, $SP_1[Obj_2/X] : \mathbf{Spec}(SP)$ is sufficient).

The other part of β-equality for objects, β-reduction, although not derivable in the system, is admissible in it[7]:

Lemma 5.1 The following rule is an admissible rule of the system

$$\frac{Obj : SP \qquad Obj \rightarrow_\beta^* Obj'}{Obj' : SP}$$

Proof (sketch) It is sufficient to consider the case $Obj \rightarrow_\beta Obj'$ (then the more general case follows by easy induction on the length of the reduction sequence). We will need an additional lemma:

Lemma 5.2 The following rule is an admissible rule of the system

$$\frac{Obj : SP \qquad \begin{matrix}[X : SP] \\ Obj' : SP'\end{matrix}}{Obj'[Obj/X] : SP'[Obj/X]}$$

Proof (idea) By obvious induction on the derivation of $Obj' : SP'$, by inspection of the rules of the system. □

The proof now is by induction on the derivation of $Obj : SP$. The only essential case is that of rule (R5) where a β-reduct may be introduced. So, in (R5) let Obj be $\lambda X{:}SP_1.\, Obj_1$, and suppose that $\lambda X{:}SP_1.\, Obj_1 : \Pi X{:}SP.\, SP'$ and $Obj' : SP$. We can assume that $\lambda X{:}SP_1.\, Obj_1 : \Pi X{:}SP.\, SP'$ has been derived using (R4): we can show that no generality is lost since (R4) is the only rule introducing λ-expressions. Hence, we have that $Obj_1 : SP'$ under the assumption $X : SP$. Thus, by Lemma 5.2, $Obj_1[Obj'/X] : SP'[Obj'/X]$, which is what we need to show. All the other cases of the inductive proof are easy; for example:

(R1): What we have to show is that whenever $Obj : SP$ and $\{Obj\} \rightarrow_\beta Obj'$ then $Obj' : \mathbf{Spec}(SP)$. Since $\{Obj\} \rightarrow_\beta Obj'$, Obj' has to be of the form $\{Obj''\}$ where $Obj \rightarrow_\beta Obj''$. By the inductive assumption, $Obj : SP$ and $Obj \rightarrow_\beta Obj''$ imply $Obj'' : SP$, and so using the same rule we derive $\{Obj''\} : \mathbf{Spec}(SP)$.

(R6): One of the assumptions of the rule is $Obj : SP$. Hence, by the inductive assumption, $Obj' : SP$, and so using the same rule we can conclude that indeed $Obj' : SP'$.

□

It might be interesting to enrich the system by the β-reduction rule for objects given in the above lemma, or even more generally by some "operational semantics rules" for (the computable part of) the object language. This, however, would be quite orthogonal to the issues of object specification considered in this paper. Therefore, to keep the system as small and as simple as possible, the rule is not included in the system.

Rules (R8) and (R9) embody trivial deductions which should be intuitively straightforward. Notice that $SP : \mathbf{Spec}(SP')$, as in the premise of (R9), asserts that specification SP imposes at least the same requirements as SP'.

[7] A rule is *admissible* in a deduction system if its conclusion is derivable in the system provided that all its premises are derivable. This holds in particular if the rule is *derivable* in the system, that is, if it can be obtained by composition of the rules in the system.

verification process which is a necessary component of inference in the above formal system. These rules are deliberately restricted to the non-parametric case, since this is the point at which an external formal system is required; parameterisation is handled by the other rules. We do not attempt here to provide a formal system for proving the semantic judgements $[\![A]\!]\rho \in [\![SP]\!]\rho$ and $[\![SP]\!]\rho \subseteq [\![SP']\!]\rho$ for all environments ρ consistent with the current context. This is an interesting and important research topic, which is however separate from the main concerns of this paper; some preliminary considerations and results on this may be found in e.g. [ST 88] and [Far 89]. It is not possible to give a set of purely "syntactic" inference rules which is sound and complete with respect to the semantics above because of the power of the specification mechanisms included in the language (this is already the case for the subset of the language excluding parameterisation, presented in Section 2).

As mentioned earlier, to make the rules as clear and readable as possible, the presentation of the system omits a full formal treatment of contexts. In particular, we should add two rules to derive judgements that a context is well-formed (here, $\langle\rangle$ is the empty context):

$$\frac{}{\langle\rangle \text{ is a well-formed context}}$$

$$\frac{\Gamma \text{ is a well-formed context} \qquad X \text{ is not in } \Gamma \qquad [\Gamma]\ SP : \mathbf{Spec}(SP_{any})}{\Gamma, X : SP \text{ is a well-formed context}}$$

and then axioms $X_1 : SP_1, \ldots, X_n : SP_n \vdash X_k : SP_k$, for $k = 1, \ldots, n$, where $X_1 : SP_1, \ldots, X_n : SP_n$ is a well-formed context. It is important to realise that contexts are *sequences*, rather than sets, and so we allow the variables $X_1, \ldots, X_k$ to occur in SP_{k+1}.

We will continue omitting contexts throughout the rest of the paper. All the definitions and facts given below (as well as above) are correctly stated for closed objects only, but are meant to be naturally extended to objects in a well-formed context. This will be done explicitly only within proofs where it is absolutely necessary. Similarly, we will omit in the following the environment argument to the semantic function for objects; all the environments thus implicitly considered are assumed to be consistent with the corresponding context. We hope that this slight informality will contribute to the readability of the paper without obscuring the details too much.

The following theorem expresses the soundness of the formal system above with respect to the semantics given earlier.

Theorem 5.3 For any object Obj and specification SP, if $Obj : SP$ is derivable then $[\![Obj]\!] \in [\![SP]\!]$ (that is, $[\![SP]\!]$ is defined and is a class of values and $[\![Obj]\!]$ is defined and is a value in this class).
Proof (sketch) By induction on the length of the derivation and by inspection of the rules. A complete formal proof requires, of course, a careful treatment of free variables and their interpretation (cf. the remark preceding the theorem). Thus, for example, rule (R4) really stands for:

$$\frac{\Gamma \vdash SP : \mathbf{Spec}(SP_{any}) \qquad \Gamma, X : SP \vdash Obj : SP' \qquad X \text{ is not in } \Gamma}{\Gamma \vdash \lambda X{:}SP.\, Obj : \Pi X{:}SP.\, SP'}$$

where Γ is a context. In the corresponding case of the inductive step we can assume that

1. $[\![SP]\!]\rho \in [\![\mathbf{Spec}(SP_{any})]\!]\rho$ for all environments ρ consistent with context Γ, and
2. $[\![Obj]\!]\rho \in [\![SP']\!]\rho$ for all environments ρ consistent with context $\Gamma, X : SP$

and then we have to prove that $[\![\lambda X{:}SP.\, Obj]\!]\rho \in [\![\Pi X{:}SP.\, SP']\!]\rho$ for all environments ρ consistent with context Γ. That is, taking into account the semantics of λ- and Π-expressions as given in Section 4, we have to prove that for all environments ρ consistent with context Γ

and then

- for all values $v \in [\![SP]\!]\rho$,
  - $[\![Obj]\!]\rho[v/X]$ is defined,
  - $[\![SP']\!]\rho[v/X]$ is defined and is a class of values, and
  - $[\![Obj]\!]\rho[v/X] \in [\![SP']\!]\rho[v/X]$,

  which in turn follow directly from assumption (2) above.

The cases corresponding to the other rules of the system require similar, straightforward but tedious analysis. Notice that the proofs about the rules concerning application and β-reduction, (R5), (R6) and (R7), crucially depend on Fact 4.1 and Corollary 4.2. □

It is natural to ask if the above formal system is also complete with respect to the semantics. It turns out not to be complete. One reason for incompleteness is that the formal system does not exploit the semantical consequences of inconsistency. For example, for any inconsistent specification SP we have that $[\![SP]\!] \in [\![\mathbf{Spec}(SP_{any})]\!]$ for any SP_{any} such that $[\![SP_{any}]\!]$ is a class of values. The corresponding formal judgement $SP : \mathbf{Spec}(SP_{any})$ is not derivable when (for example) SP and SP_{any} are simple specifications over different signatures. If the formal parameter specification in a λ- or Π-expression is inconsistent then similar difficulties arise (cf. [MMMS 87] for a discussion of the related issue of "empty types" in typed λ-calculi). This topic deserves further study; it might be that the system is complete when inconsistencies are excluded and perhaps some additional restrictions on the objects and specifications involved are imposed (although the deliberate omission of a rule allowing for well-formed β-expansion of objects makes this unlikely).

Definition 5.4 An object Obj is *well-formed* if $Obj : SP$ for some SP. □

This also defines the well-formed specifications since specifications are objects.

Checking whether an expression in the language is well-formed must in general involve "semantic" verification as embodied in rules (R10) and (R11). In fact, checking the well-formedness of objects is as hard as checking if they satisfy specifications: $Obj : SP$ if and only if $(\lambda X{:}SP.\ (\text{any constant}))(Obj)$ is well-formed.

An easy corollary to the soundness theorem is the following:

Corollary 5.5 Any well-formed object Obj has a well-defined meaning $[\![Obj]\!]$. □

Since specifications do not form a separate syntactic category of the language, in the above discussion we have used the term "specification" and the meta-variable SP rather informally, relying on an intuitive understanding of the role of the objects of the language. This intuitive understanding may be made formal as follows:

Definition 5.6 An object SP is called a *specification* if for some SP_{any}, $SP : \mathbf{Spec}(SP_{any})$. □

Corollary 5.7 The meaning of a specification is a class of values: if $SP : \mathbf{Spec}(SP_{any})$ then $[\![SP]\!] \subseteq [\![SP_{any}]\!]$. □

Note that this covers ordinary Σ-specifications, specifications of (higher-order) parametric algebras, specifications of (higher-order) parameterised specifications, etc. The following theorem shows that this is indeed consistent with our previous informal use of the term.

Theorem 5.8 If $Obj : SP$ then SP is a specification.

Proof We prove that $SP : \mathbf{Spec}(SP_{any})$ for some SP_{any} by induction on the derivation of $Obj : SP$, by inspection of the rules of the system:

$\mathbf{Spec}(\Sigma)$ is indeed a specification as we have $\mathbf{Spec}(\Sigma) : \mathbf{Spec}(\mathbf{Spec}(\Sigma))$, which may be derived by using the rule introducing Σ, and then the rule of $\mathbf{Spec}(\_)$-introduction (R3).

(R1): By the inductive assumption we have $SP : \mathbf{Spec}(SP_{any})$, from which we can derive $\mathbf{Spec}(SP) : \mathbf{Spec}(\mathbf{Spec}(SP_{any}))$.

(R2): We need the following lemma:

> **Lemma 5.9** If an object $\mathbf{Spec}(SP)$ is well-formed then SP is a specification.
>
> **Proof** We proceed by induction on a derivation of the well-formedness of $\mathbf{Spec}(SP)$, by inspection of the possible last rules in the derivation:
>
> (R3): Clearly, we have here $SP : \mathbf{Spec}(SP')$ as the assumption for the use of this rule.
>
> (R6), (R7), (R8), (R9): Let Obj be $\mathbf{Spec}(SP)$. One of the premises of each of these rules implies the well-formedness of $\mathbf{Spec}(SP)$ and so the inductive assumption implies that SP is a specification.
>
> (R11): As in the previous case, but take SP to be $\mathbf{Spec}(SP)$. (In fact, this case is vacuous since $\mathbf{Spec}(SP) : \mathbf{Spec}(\Sigma)$ is not derivable anyway.)
>
> Notice that only the first case of the above was essential: it is sufficient to analyse only the rules that may be used to "build" objects of the form we consider (the $\mathbf{Spec}(\_)$-introduction rule (R3) in this case). We have relied on a similar remark in the proof of Lemma 5.1. □

By the inductive assumption (of the proof of the theorem) we have that under the assumption $X : SP$, $\mathbf{Spec}(SP'')$ is well-formed, and so using the above lemma we conclude that $SP'' : \mathbf{Spec}(SP'_{any})$. Hence, we can derive $\Pi X{:}SP.\, SP'' : \mathbf{Spec}(\Pi X{:}SP.\, SP'_{any})$, and then $\mathbf{Spec}(\Pi X{:}SP.\, SP'') : \mathbf{Spec}(\mathbf{Spec}(\Pi X{:}SP.\, SP'_{any}))$.

(R3): By the inductive assumption, $\mathbf{Spec}(SP') : \mathbf{Spec}(SP_{any})$, which entails $\mathbf{Spec}(\mathbf{Spec}(SP')) : \mathbf{Spec}(\mathbf{Spec}(SP_{any}))$.

(R4): By the inductive assumption we have that under the assumption $X : SP$, $SP' : \mathbf{Spec}(SP'_{any})$, and so we can derive $\Pi X{:}SP.\, SP' : \mathbf{Spec}(\Pi X{:}SP.\, SP'_{any})$.

(R5): The inductive assumption implies that $\Pi X{:}SP.\, SP'$ is well-formed. We prove that this implies that SP' is a specification under the assumption $X : SP$. The proof is by induction on the derivation of the well-formedness of $\Pi X{:}SP.\, SP'$, by inspection of the possible last rules used in the derivation. As in the proof of Lemma 5.9, it is sufficient to analyse the Π-introduction rule (R2). Since what we need is one of the assumptions for the applicability of this rule, we can indeed conclude that $SP' : \mathbf{Spec}(SP'')$ under the assumption $X : SP$. Hence, by Lemma 5.2 we conclude that $SP'[Obj'/X] : \mathbf{Spec}(SP''[Obj'/X])$.

(R6): By the inductive assumption applied to one of the premises of this rule, SP is a specification. Thus, since $SP \rightarrow^*_\beta SP'$, by Lemma 5.1 it follows that SP' is a specification as well.

(R7): Trivial.

(R8): From the premise of the rule, we can directly derive $\{Obj\} : \mathbf{Spec}(SP_{any})$.

(R9): By the inductive assumption, from the premise of the rule it follows that $\mathbf{Spec}(SP')$ is well-formed. Thus, by Lemma 5.9, SP' is a specification.

(R10): Trivial.

(R11): From the premise $SP' : \mathbf{Spec}(\Sigma)$, we derive $\mathbf{Spec}(SP') : \mathbf{Spec}(\mathbf{Spec}(\Sigma))$.

It is perhaps surprising how long and relatively complicated the proof of an intuitively rather obvious fact has become here. Unfortunately, this seems to be typical of many proofs dealing with "syntactic" properties of λ-calculi.

6 Type-checking

Inference in the system presented in the previous section has a purely "type-checking" component on which the "verification" component is in a sense superimposed. We try to separate this "type-checking" process below. The concept of type we use must cover signatures (as "basic types" of algebras) and "arrow types" (types of functions) which would be usual in any type theory, as well as "specification types" which are particular to the formalism presented here: as we have stressed before, the type of a specification is distinct from the type of objects the specification specifies.

Definition 6.1 The class of types $\mathcal{T}$ is defined as the least class such that:

- for any signature Σ, $\Sigma \in \mathcal{T}$;
- for any types $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \rightarrow \tau_2 \in \mathcal{T}$; and
- for any type $\tau \in \mathcal{T}$, $\mathbf{Spec}(\tau) \in \mathcal{T}$.

□

Under the standard notational convention that arrow types of the form $\tau \rightarrow \tau'$ stand for Π-types of the form $\Pi X{:}\tau.\, \tau'$ where X does not actually occur in τ', types as defined above are well-formed specifications.

We define type $Type(Obj)$ for an object Obj of our system by induction as follows:

Simple specifications:

$$\frac{\Sigma \text{ signature}}{Type(\Sigma) = \mathbf{Spec}(\Sigma)} \qquad \frac{Type(SP) = \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{Type(\mathbf{impose}\ \Phi\ \mathbf{on}\ SP) = \mathbf{Spec}(\Sigma)}$$

... and similarly for other simple specifications ...

Other specifications:

$$\frac{Type(Obj) = \tau}{Type(\{Obj\}) = \mathbf{Spec}(\tau)} \qquad \frac{Type(SP) = \mathbf{Spec}(\tau)}{Type(\mathbf{Spec}(SP)) = \mathbf{Spec}(\mathbf{Spec}(\tau))}$$

$$\frac{Type(SP) = \mathbf{Spec}(\tau) \qquad \begin{matrix}[Type(X) = \tau] \\ Type(SP') = \mathbf{Spec}(\tau')\end{matrix}}{Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(\tau \rightarrow \tau')}$$

λ-expressions:

$$\frac{Type(SP) = \mathbf{Spec}(\tau) \qquad \begin{matrix}[Type(X) = \tau] \\ Type(Obj) = \tau'\end{matrix}}{Type(\lambda X{:}SP.\, Obj) = \tau \rightarrow \tau'} \qquad \frac{Type(Obj) = \tau \rightarrow \tau' \qquad Type(Obj') = \tau}{Type(Obj(Obj')) = \tau'}$$

Algebra expressions:

$$\frac{A \text{ is an algebra expression denoting a } \Sigma\text{-algebra}}{Type(A) = \Sigma}$$

and the β-reduction and β-expansion rules (R6) and (R7), which do not introduce new well-formed objects, do not have counterparts in the above definition.

Clearly, the above definition depends on a judgement whether or not an algebra expression denotes an algebra over a given signature. We will assume that such "type-checking" of algebra expressions is defined externally in such a way that it is consistent with the semantics (i.e., if A is a well-formed algebra expression denoting a Σ-algebra then indeed $[\![A]\!] \in Alg(\Sigma)$). Moreover, we will assume that it is substitutive: if A is an algebra expression denoting a Σ-algebra under an assumption $Type(X) = \tau$ then for any object Obj with $Type(Obj) = \tau$, $A[Obj/X]$ is an algebra expression denoting a Σ-algebra as well.

The above rules (deliberately) do not define $Type(Obj)$ for all object expressions of our language. However, if a type is defined for an object, it is defined unambigously. An object Obj is *roughly well-formed* if its type $Type(Obj)$ is defined. There are, of course, roughly well-formed objects that are not well-formed. The opposite implication holds, though:

Theorem 6.2 $Type(Obj)$ is well-defined for any well-formed object Obj. In particular:

1. If $Obj : SP$ then $Type(SP) = \mathbf{Spec}(Type(Obj))$.
2. If SP is a specification then $Type(SP) = \mathbf{Spec}(\tau)$ for some type τ.
3. If $Obj : \Pi X{:}SP.\, SP'$ then $Type(Obj) = \tau \rightarrow \tau'$, where $Type(SP) = \mathbf{Spec}(\tau)$, for some types τ and τ'.

Proof The first part of the theorem follows by induction on the length of the derivation (we sketch this proof below). The other two parts follow directly from this.

Let us first rephrase the first part of the theorem taking contexts describing free variables explicitly into account, which is perhaps not entirely obvious here:

1′. If $Obj : SP$ is derivable under assumptions $X_1 : SP_1, \ldots, X_n : SP_n$ where $Type(SP_1) = \mathbf{Spec}(\tau_1)$, ..., $Type(SP_n) = \mathbf{Spec}(\tau_n)$, then $Type(SP) = \mathbf{Spec}(Type(Obj))$ under the assumptions $Type(X_1) = \tau_1$, ..., $Type(X_n) = \tau_n$.

Now, we prove this part of the theorem by induction on the derivation of $Obj : SP$, by inspection of the rules:

Simple specifications: The rules for simple specifications cause no problem, since using the inductive assumption we conclude that each well-formed specification SP in the conclusion of these rules has type $Type(SP) = \mathbf{Spec}(\Sigma)$, and $Type(\mathbf{Spec}(\Sigma)) = \mathbf{Spec}(\mathbf{Spec}(\Sigma))$.

(R1): By the inductive assumption we have $Type(SP) = \mathbf{Spec}(Type(Obj))$, hence $Type(\mathbf{Spec}(SP)) = \mathbf{Spec}(Type(SP)) = \mathbf{Spec}(\mathbf{Spec}(Type(Obj))) = \mathbf{Spec}(Type(\{Obj\}))$.

(R2): We need the following lemma:

> **Lemma 6.3** If $\mathbf{Spec}(SP)$ has a type then SP has a type of the form $\mathbf{Spec}(\tau)$.
>
> **Proof** Obvious, since the only way to derive a type for $\mathbf{Spec}(SP)$ is using the rule
>
> $$\frac{Type(SP) = \mathbf{Spec}(\tau)}{Type(\mathbf{Spec}(SP)) = \mathbf{Spec}(\mathbf{Spec}(\tau))}$$
>
> which requires that indeed $Type(SP) = \mathbf{Spec}(\tau)$ for some type τ. □

where $Type(SP) = \mathbf{Spec}(\tau)$, $\mathbf{Spec}(SP'')$ has a type, and so using the above lemma we conclude that $Type(SP'') = \mathbf{Spec}(\tau'')$ for some type τ''. Hence, we can derive $Type(\Pi X{:}SP.\, SP'') = \mathbf{Spec}(\tau{\rightarrow}\tau'')$, and then $Type(\mathbf{Spec}(\Pi X{:}SP.\, SP'')) = \mathbf{Spec}(\mathbf{Spec}(\tau{\rightarrow}\tau''))$.

On the other hand, by the inductive assumption again, under the assumption $Type(X) = \tau$, $Type(\mathbf{Spec}(SP'')) = \mathbf{Spec}(Type(SP'))$. Hence, $\mathbf{Spec}(\mathbf{Spec}(\tau'')) = \mathbf{Spec}(Type(SP'))$, and so $Type(SP') = \mathbf{Spec}(\tau'')$. Thus, $Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(\tau{\rightarrow}\tau'')$, which completes the proof in this case.

(R3): By the inductive assumption, $\mathbf{Spec}(Type(SP)) = Type(\mathbf{Spec}(SP'))$, which easily implies $\mathbf{Spec}(Type(\mathbf{Spec}(SP))) = Type(\mathbf{Spec}(\mathbf{Spec}(SP')))$.

(R4): By the inductive assumption, using Lemma 6.3, we have that $Type(SP) = \mathbf{Spec}(\tau)$ for some type τ, and then under the assumption $Type(X) = \tau$, $Type(SP') = \mathbf{Spec}(\tau')$ where $\tau' = Type(Obj)$. Thus, we can derive both $Type(\lambda X{:}SP.\, Obj) = \tau{\rightarrow}\tau'$ and $Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(\tau{\rightarrow}\tau')$

(R5): The inductive assumption implies that $Type(SP) = \mathbf{Spec}(\tau)$ where $\tau = Type(Obj')$, and that $Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(Type(Obj))$. Since there is only one rule which allows us to derive a type for the Π-expression, by a direct analysis of this rule we can conclude that under the assumption $Type(X) = \tau$, $Type(SP') = \mathbf{Spec}(\tau')$ for some type τ'. Moreover, $Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(\tau{\rightarrow}\tau')$, which implies $Type(Obj) = \tau{\rightarrow}\tau'$. Hence, we can derive $Type(Obj(Obj')) = \tau'$.

> **Lemma 6.4** For any object Obj', variable X and type τ, if $Type(Obj') = \tau'$ under the assumption $Type(X) = \tau$, then for any object Obj such that $Type(Obj) = \tau$, we have $Type(Obj'[Obj/X]) = \tau'$.
>
> **Proof** By obvious induction on the derivation of $Type(Obj') = \tau'$, by inspection of the clauses in the definition of $Type(\_)$. □

Hence, by the above lemma we conclude that $Type(SP'[Obj'/X]) = \mathbf{Spec}(\tau')$, which completes the proof in this case.

(R6): We need the following lemma:

> **Lemma 6.5** β-reduction preserves types of objects. That is, for any object Obj such that $Type(Obj) = \tau$, if $Obj \rightarrow^*_\beta Obj'$ then $Type(Obj') = \tau$.
>
> **Proof (sketch)** It is sufficient to show the lemma for $Obj \rightarrow_\beta Obj'$. The proof proceeds by induction on the derivation of the type of Obj. The only non-trivial case is that of application, where a β-reduct may be introduced. So, assume that Obj is a roughly well-formed object of the form $(\lambda X{:}SP.\, Obj_1)(Obj_2)$. Then, for some types τ and τ', $Type((\lambda X{:}SP.\, Obj_1)(Obj_2)) = \tau'$, $Type(\lambda X{:}SP.\, Obj_1) = \tau{\rightarrow}\tau'$, $Type(Obj_2) = \tau$, $Type(SP) = \mathbf{Spec}(\tau)$ and under the assumption $Type(X) = \tau$, $Type(Obj_1) = \tau'$. Hence, by Lemma 6.4, $Type(Obj_1[Obj_2/X]) = \tau'$, which is what is needed in this case. □

Now, by the inductive assumption applied to one of the premises of the rule we have that $\mathbf{Spec}(Type(Obj)) = Type(SP)$. Then, since $SP \rightarrow^*_\beta SP'$, by the above lemma we have indeed $\mathbf{Spec}(Type(Obj)) = Type(SP')$.

(R7): Similarly as in the previous case.

(R8), (R9), (R10), (R11): Easy use of the inductive assumption.

□

both are roughly well-formed and the type of the object is consistent with the type of the specification. Of course, nothing like the opposite implications holds. As pointed out earlier, proving that an object satisfies a specification must involve a verification process as embodied in the two rules of semantic inference.

One might now expect that any well-formed object Obj "is of its type", i.e. $Obj : Type(Obj)$. This is not the case, though. The problem is that both λ- and Π-expressions include parameter *specifications* rather than just parameter *types*, and so functions denoted by λ-expressions and specified by Π-expressions have domains defined by specifications, not just by types. This is necessary for methodological reasons: we have to be able to specify permissible arguments in a more refined way than just by giving their types. However, as a consequence, objects denoted by λ- and Π-expressions in general do not belong to the domain defined by their types, and so we cannot expect that such expressions would "typecheck" to their types.

To identify the purely "type-checking" component in our system we have to deal with objects where parameter specifications are replaced by their types. Formally, for any roughly well-formed object Obj, its version $Erase(Obj)$ with parameter specifications erased is defined by induction as follows:

Specifications:

$$\begin{array}{rcl} Erase(\Sigma) & =_{def} & \Sigma \\ Erase(\textbf{impose } \Phi \textbf{ on } SP) & =_{def} & \textbf{impose } \Phi \textbf{ on } Erase(SP) \end{array}$$

... and similarly for other simple specifications ...

$$\begin{array}{rcl} Erase(\{Obj\}) & =_{def} & \{Erase(Obj)\} \\ Erase(\Pi X{:}SP.\, SP') & =_{def} & \Pi X{:}\tau.\, Erase(SP') \\ & & \quad \text{where } Type(SP) = \textbf{Spec}(\tau) \\ Erase(\textbf{Spec}(SP)) & =_{def} & \textbf{Spec}(Erase(SP)) \end{array}$$

Other objects:

$$\begin{array}{rcl} Erase(A) & =_{def} & A \\ Erase(\lambda X{:}SP.\, Obj) & =_{def} & \lambda X{:}\tau.\, Erase(Obj) \\ & & \quad \text{where } Type(SP) = \textbf{Spec}(\tau) \\ Erase(Obj(Obj')) & =_{def} & Erase(Obj)(Erase(Obj')) \end{array}$$

We have chosen here to define $Erase(A) = A$ for all algebra expressions A. Alternatively, we could leave this case out again, and require a definition to be provided externally. For example, one might want that $Erase(A[Obj/X]) = Erase(A[Erase(Obj)/X])$ (which would not necessarily hold under the above definition). The only property we need is that if A is an algebra expression denoting a Σ-algebra then so is $Erase(A)$.

Theorem 6.6 For any roughly well-formed object Obj, $Erase(Obj) : Type(Obj)$ (hence, $Erase(Obj)$ is well-formed).

Proof (idea) Again, the extension to objects with free variables is not entirely clear. What we mean is: if $Type(Obj) = \tau$ under the assumptions $Type(X_1) = \tau_1$, ..., $Type(X_n) = \tau_n$ then $Erase(Obj) : Type(Obj)$ under the assumptions $X_1 : \tau_1, \ldots, X_n : \tau_n$. This may be proved by straightforward induction on the derivation of the type of Obj. □

Joining this with Theorem 6.2, we conclude that a necessary condition for an object to satisfy a specification is that the version of the object where parameter specifications have been "rounded up" to parameter types has a type which is consistent with the type of the specification. This necessary condition embodies the purely type-checking component of any proof that an object satisfies a specification.

Proof This follows directly from Theorems 6.6 and 6.2 since for any type τ, $Type(\tau) = \mathbf{Spec}(\tau)$, which may easily be established by an obvious induction on the structure of types. □

The above corollary, when the equality is read from right to left, may be viewed as an alternative definition of the type of a roughly well-formed object. The type-checking of $Erase(Obj)$ may be performed within the original system separately from the semantic verification part, without any reference to the meanings of objects and specifications. We present below the corresponding proper fragment of the original system:

Simple specifications:

$$\frac{\Sigma \text{ signature}}{\Sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{\mathbf{impose}\ \Phi\ \mathbf{on}\ SP : \mathbf{Spec}(\Sigma)}$$

... and just as before for other simple specifications ...

Other specifications:

$$\frac{Obj : \tau}{\{Obj\} : \mathbf{Spec}(\tau)} \qquad \frac{\begin{array}{c}[X : \tau] \\ SP' : \mathbf{Spec}(\tau')\end{array}}{\Pi X{:}\tau.\ SP' : \mathbf{Spec}(\tau \rightarrow \tau')} \qquad \frac{SP : \mathbf{Spec}(\tau)}{\mathbf{Spec}(SP) : \mathbf{Spec}(\mathbf{Spec}(\tau))}$$

λ-expressions:

$$\frac{\begin{array}{c}[X : \tau] \\ Obj : \tau'\end{array}}{\lambda X{:}\tau.\ Obj : \tau \rightarrow \tau'} \qquad \frac{Obj : \tau \rightarrow \tau' \qquad Obj' : \tau}{Obj(Obj') : \tau'}$$

Algebra expressions:

$$\frac{A \text{ is an algebra expression denoting a } \Sigma\text{-algebra}}{A : \Sigma}$$

We hope that a comparison of the above with the system presented in Section 5 should clearly illustrate the intuitive difference between typed λ-calculi, like the one above, and "specified" λ-calculi, like the one in Section 5.

7 Concluding remarks

Spurred by the methodological considerations in [SST 90], we have presented an institution-independent specification formalism which provides a notation for parameterised specifications and specifications of parametric objects of an arbitrary order, as well as any mixture of these concepts. The formalism incorporates the kernel specification-building operations described in [ST 88] based on those in the ASL specification language [SW 83], [Wir 86]. The basic idea was to treat specifications, which specify objects, as objects themselves. This collapsing together of the two levels, that of objects and that of their specifications, led (perhaps surprisingly) to a well-behaved inference system for proving that an object satisfies a specification with a clearly identified formal type-checking component (cf. [SdST 90] where the formal type-checking component of Extended ML is given).

The formalism presented deals explicitly with two levels of objects involved in the process of software development: programs (viewed as algebras) and their specifications (viewed as classes of algebras) — both, of course, arbitrarily parameterised. Aiming at the development of an institution-independent framework, we decided to omit from our considerations yet another level of objects

particular institutions, however, it may be interesting to explicitly consider this level as well, and to intermix constructs for dealing with this level with those for the other two levels mentioned above. This would lead to entities such as algebras parametric on data values, specifications parameterised by functions on data, functions from algebras and specifications to data values, etc.

Just as the kernel ASL-like specification formalism it builds on, the presented system is too low-level to be directly useful in practice. We view it primarily as a kernel to be used as a semantic foundation for the development of more user-friendly specification languages. An example of such a more user-oriented framework is the Extended ML specification language [ST 85] which comes together with a program development methodology as presented in [ST 89]. The formalism described in this paper provides adequate foundations for Extended ML. Indeed, one of the main stimuli for its development was our inability to express the semantics of the current version of Extended ML directly in terms of the kernel specification-building operations in ASL: Extended ML functor specifications are specifications of parametric objects, and these were not present in ASL. The task of writing out a complete institution-independent semantics of Extended ML in terms of the specification formalism presented here remains to be done. We expect that some technicalities, like those which arise in connection with ML type inheritance, will cause the same problems as in [ST 89]. Some others, like the use of behavioural equivalence and the concept of functor stability in the Extended ML methodology, although directly related to the **abstract** operation in the formalism presented here, require further study in this more general framework. Finally, properties of ML functors such as persistency, which cause difficulties in other specification formalisms, will be easy to express here.

Of course, the formal properties of the system need much further study. For example, it seems that the "cut" rule should be admissible (although not derivable) in the remainder of the system. The standard properties of β-reduction, such as the Church-Rosser property and termination (on well-formed objects) should be carefully proven, probably by reference to the analogous properties of the usual typed λ-calculus. For example, the termination property of β-reduction on the well-formed objects of the language should follow easily from the observation that the *Erase* function as defined in Section 6 preserves β-reduction, which allows us to lift the corresponding property of the usual typed λ-calculus to our formalism. The system is incomplete, as pointed out earlier. It would be useful to identify all the sources of this incompleteness, for example by characterising an interesting subset of the language for which the system is complete. One line of research which we have not followed (as yet) is to try to encode the formalism we present here in one of the known type theories (for example, Martin-Löf's system [NPS 90], the calculus of constructions [CH 88] or LF [HHP 87]). It would be interesting to see both which of the features of the formalism we propose would be difficult to handle, as well as which of the tedious proofs of some formal properties of our formalism (cf. the proofs sketched for Theorems 5.8 and 6.2) would turn out to be available for free under such an encoding.

Acknowledgements: We are grateful to Stefan Sokołowski for his collaboration on [SST 90] leading to the development of an early version of the formalism presented here. Thanks to Jordi Farrés, Cliff Jones and Stefan Kahrs for helpful comments on a draft of [SST 90], and to Jan Bergstra for a question on β-reduction which led to the current version of the system. Thanks to an anonymous referee for comments on a draft of this paper which helped to improve the presentation.

This research was supported by the University of Edinburgh, the University of Bremen, the Technical University of Denmark, the University of Manchester, and by grants from the Polish Academy of Sciences, the (U.K.) Science and Engineering Research Council, ESPRIT, and the Wolfson Foundation.

8 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[Ada 80] *The Programming Language Ada: Reference Manual.* LNCS 106 (1980).

Holland (1984).

[Bau 85] F.L. Bauer *et al* (the CIP language group). *The Wide Spectrum Language CIP-L.* LNCS 183 (1985).

[BGM 89] M. Bidoit, M.-C. Gaudel and A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. *Science of Computer Programming* 12, 1–38 (1989).

[BT 83] A. Blikle and A. Tarlecki. Naive denotational semantics. *Information Processing 83, Proc. IFIP Congress '83* (ed. R. Mason), Paris. North-Holland, 345–355 (1983).

[BKS 88] A.M. Borzyszkowski, R. Kubiak and S. Sokołowski. A set-theoretic model for a typed polymorphic λ-calculus. *Proc. VDM-Europe Symp. VDM — The Way Ahead*, Dublin. LNCS 328, 267–298 (1988).

[BG 80] R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. *Proc. of Advanced Course on Abstract Software Specification*, Copenhagen. LNCS 86, 292–332 (1980).

[Coh 81] P.M. Cohn. *Universal Algebra.* Reidel (1981).

[Con 86] R.L. Constable *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall (1986).

[CH 88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation* 76 (1988).

[DMN 70] O.-J. Dahl, B. Myrhaug and K. Nygaard. Simula 67 common base language. Report S-22, Norwegian Computing Center, Oslo (1970).

[Ehr 82] H.-D. Ehrich. On the theory of specification, implementation, and parametrization of abstract data types. *Journal of the Assoc. for Computing Machinery 29*, 206–227 (1982).

[EKTWW 84] H. Ehrig, H.-J. Kreowski, J. Thatcher, E. Wagner and J. Wright. Parameter passing in algebraic specification languages. *Theoretical Computer Science* 28, 45–81 (1984).

[EM 85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer (1985).

[ETLZ 82] H. Ehrig, J.W. Thatcher, P. Lucas and S.N. Zilles. Denotational and initial algebra semantics of the algebraic specification language LOOK. Report 84-22, Technische Universität Berlin (1982).

[Far 89] J. Farrés-Casals. Proving correctness of constructor implementations. *Proc. 14th Symp. on Mathematical Foundations of Computer Science*, Porąbka-Kozubnik. LNCS 379, 225–235 (1989).

[FJKR 87] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans and G.R. Renardel de Lavalette. Formal definition of the design language COLD-K. METEOR Report t7/PRLE/7, Philips Research Laboratories (1987).

[FJ 90] J.S. Fitzgerald and C.B. Jones. Modularizing the formal description of a database system. *Proc. VDM'90 Symp. VDM and Z — Formal Methods in Software Development*, Kiel. LNCS 428, 189–210 (1990).

[Gog 84] J.A. Goguen. Parameterized programming. *IEEE Trans. Software Engineering SE-10*, 528–543 (1984).

[GB 84] J.A. Goguen and R.M. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. LNCS 164, 221–256 (1984).

[GHW 85] J.V. Guttag, J.J. Horning and J. Wing. Larch in five easy pieces. Report 5, DEC Systems Research Center, Palo Alto, CA (1985).

Symp. on Logic in Computer Science, Cornell, 194–204 (1987).

[LL 88] T. Lehmann and J. Loeckx. The specification language of OBSCURE. *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland. LNCS 332, 131–153 (1988).

[Lis 81] B.H. Liskov *et al. CLU Reference Manual.* LNCS 114 (1981).

[MacQ 86] D.B. MacQueen. Modules for Standard ML. In: R. Harper, D.B. MacQueen and R. Milner. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MMMS 87] A.R. Meyer, J.C. Mitchell, E. Moggi and R. Statman. Empty types in polymorphic lambda calculus. *Proc. 14th ACM Symp. on Principles of Programming Languages*, 253–262; revised version in *Logical Foundations of Functional Programming* (ed. G. Huet), Addison-Wesley, 273–284 (1990).

[MTH 90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[Mos 89a] P. Mosses. Unified algebras and modules. *Proc. 16th ACM Symp. on Principles of Programming Languages*, Austin, 329–343 (1989).

[Mos 89b] P. Mosses. Unified algebras and institutions. *Proc. 4th IEEE Symp. on Logic in Computer Science*, Asilomar, 304–312 (1989).

[NPS 90] B. Nordström, K. Petersson and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford Univ. Press (1990).

[SdST 90] D. Sannella, F. da Silva and A. Tarlecki. Syntax, typechecking and dynamic semantics for Extended ML (version 2). Draft report, Univ. of Edinburgh (1990). Version 1 appeared as Report ECS-LFCS-89-101, Univ. of Edinburgh (1989).

[SST 90] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. Report 6/90, Informatik, Universität Bremen (1990).

[ST 85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 67–77 (1985).

[ST 87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *J. Comp. and Sys. Sciences 34*, 150–178 (1987).

[ST 88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation 76*, 165–210 (1988).

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. Report ECS-LFCS-89-71, Univ. of Edinburgh (1989); extended abstract in *Proc. Colloq. on Current Issues in Programming Languages*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Barcelona. LNCS 352, 375–389 (1989).

[SW 83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. LNCS 158, 413–427 (1983).

[Sch 86] O. Schoett. Data abstraction and the correctness of modular programming. Ph.D. thesis, Univ. of Edinburgh (1986).

[Voß 85] A. Voß. Algebraic specifications in an integrated software development and verification system. Ph.D. thesis, Universität Kaiserslautern (1985).

[Wir 86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42, 123–249 (1986).

[Wirth 88] N. Wirth. *Programming in Modula-2* (third edition). Springer (1988).